Message Blaster: Processing Messages in Visual Basic

Ed Staffin, Microsoft Consulting Services
Kyle Marsh, Microsoft Developer Network Technology Group

Created: April 30, 1993

Abstract

The Microsoft® Visual Basic™ development environment is not based on a message-driven programming model. Instead, Visual Basic supports a predefined set of events for each object (form or control) that you create. An application written in Visual Basic cannot respond to messages from Microsoft Windows™ that are not handled directly by a Visual Basic event. The Message Blaster is a Visual Basic control that addresses this restriction by allowing you to catch and process Windows messages from Visual Basic. This article describes how the Message Blaster works.

Introduction

Visual Basic™ is a powerful and easy-to-use programming environment for Microsoft® Windows™. Unlike C developers, Visual Basic developers can create Windows-based applications without using the message-driven programming model. In Visual Basic, the developer can place a control on a form, double-click the control, select an event, and enter the basic code that is executed when the control receives that event from Windows. The complications of how that code gets executed are hidden from the developer. This is one of the features that makes Visual Basic so easy to use.

On the downside, Visual Basic does not provide a way for an application to respond to events that are not built into Visual Basic. For example, your Visual Basic application cannot track the currently selected command in a menu to display its description in the status bar. The inability to handle arbitrary events limits the functionality of Visual Basic applications. Ed Staffin of Microsoft Consulting Services created a Visual Basic control called the Message Blaster to help the developer work around this limitation. The Message Blaster allows an application to respond to events that are not built into Visual Basic.

You can use the Message Blaster control to process any message from Windows except for WM_CREATE and WM_NCCREATE messages. For example, you can process:

- WM_MENUINIT and WM_MENUSELECT messages, to change status bar text as the user moves through menu items. The EX1 sample application demonstrates this feature.
- WM_NC* (nonclient) messages, to create captions. The SmallCap sample application (created by Randall Kern and Jim Cash) does just that.
- WM_DROPFILE messages, to implement a drag-and-drop interface.

What's Going On Here?

Windows notifies applications of events by sending them messages. Applications written in C always have a message loop that receives the messages from Windows and dispatches them to the appropriate window. Each window has a window procedure that processes all messages the window receives. (See Figure 1.)

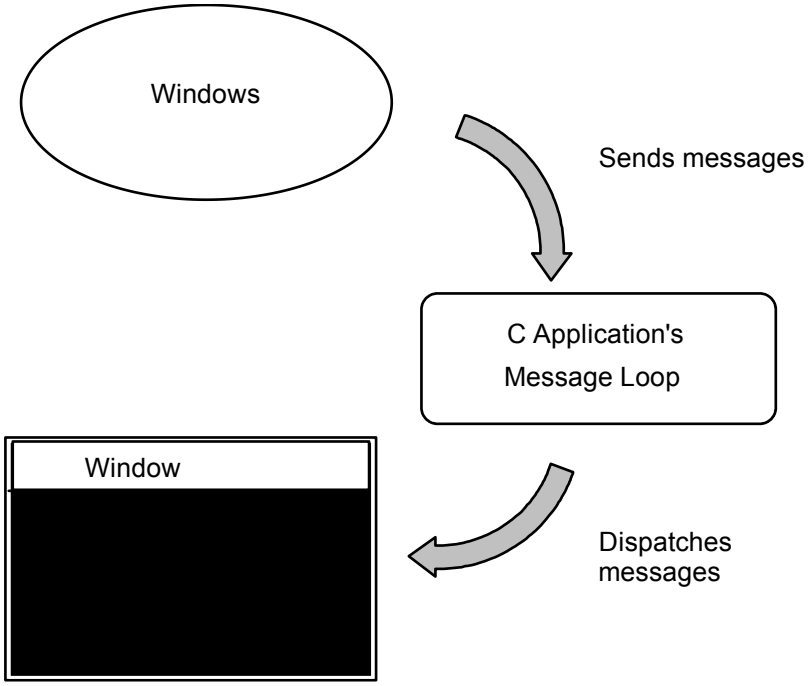

Figure 1. Message processing for a C application

The procedure is essentially the same for Visual Basic applications. However, Visual Basic supplies the message loop, and the Visual Basic controls (either built-in or custom controls) generate events for the Visual Basic application to process. (See Figure 2.)

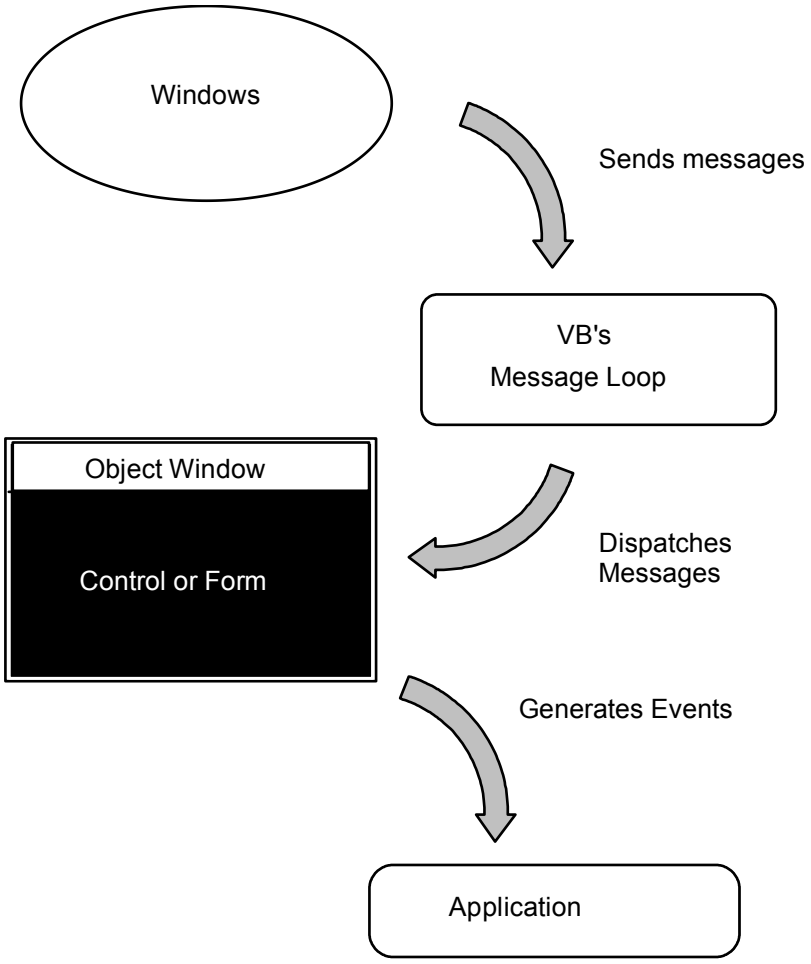

Figure 2. Processing messages in Visual Basic

Applications developed in Visual Basic must also be able to receive and process messages from Windows to handle these events. The Message Blaster intercepts the messages for a particular window (that is, a form or control), and generates an event for the Message Blaster control, which the Visual Basic application can then handle. (See Figure 3.)

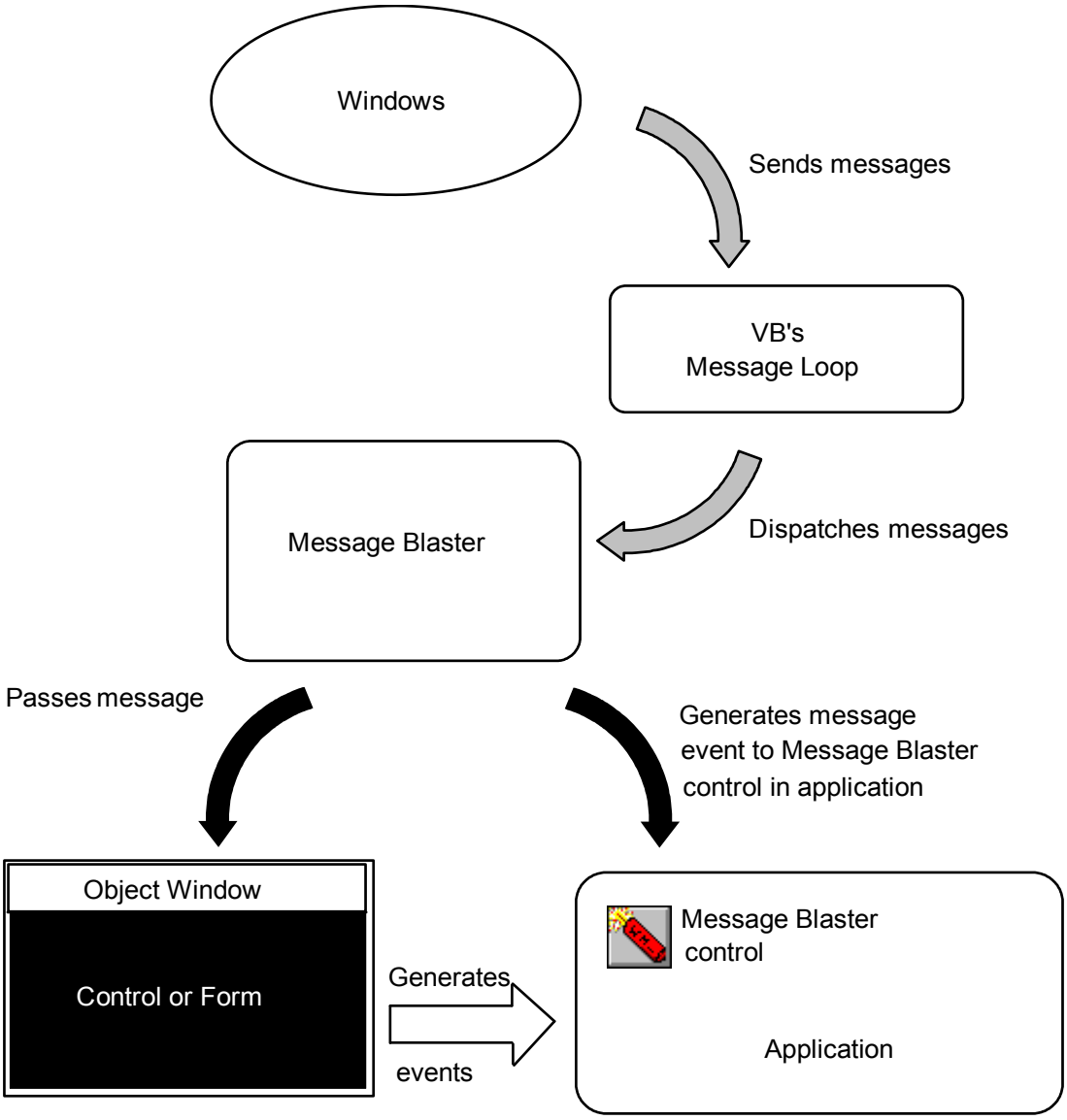

Figure 3. Message Blaster in action

Using the Message Blaster

To use the Message Blaster, follow the steps below.

1. Add the MSGBLAST.VBX file to your project. This file will add a Message Blaster icon (illustrated below) to your Visual Basic tool palette.

2. Draw a Message Blaster control on a form. Like the timer control, the Message Blaster control is visible on the form but does not appear in your application's interface. (If your form contains multiple Message Blaster controls, they will appear stacked in the upper-left corner the next time you open the form.)

3. In the **Form_Load** event procedure for the form, specify the window handle of the target object whose messages you want to process. The target object is usually the form itself or a control on the form. The easiest way to get the window handle is to use the *hWnd* property of the object. A Message Blaster can monitor only one object at a time, but you can change the target object at run time by resetting the *hWndTarget* property.

4. Specify the messages to capture in the **Form_Load** event procedure for the form. Message Blaster can handle all messages except WM_CREATE and WM_NCCREATE, which are sent before the object is created (an application cannot pass the window handle of an object that doesn't exist to Message Blaster). Assign the message numbers to be captured to the elements of the *MsgList* property array. The MESSAGES.TXT file included with the Message Blaster contains all Windows messages in Visual Basic format.

   You can also use a Message Blaster for application-defined messages. Each Message Blaster control can handle up to 25 messages. Here is a sample **Form_Load** event procedure that initializes a Message Blaster control:

```
Sub Form_Load ()
    MsgBlaster1.hWndTarget = text1.hWnd
    MsgBlaster1.MsgList(0) = WM_NCHITTEST
End Sub
```

5. Add code to handle the messages when the application receives a Message Blaster event. Each Message Blaster triggers a **Message** event when it detects a message. The **Message** event supplies four parameters to the Visual Basic application:

   - The *MsgVal* parameter, which supplies the message number.
   - The *wParam* of the message.
   - The *lParam* of the message.
   - The *ReturnVal* parameter, which the Visual Basic application can use to return a value to Windows in response to the given message.

   The values of the *wParam* and *lParam* parameters depend on the specific message. These values are documented in the Windows version 3.1 Software Development Kit (SDK), which you can find under Product Documentation on the Microsoft Developer Network CD.

   The three message handlers for the EX1 sample application are shown below.

```
Sub MsgBlaster1_Message (MsgVal As Integer, wParam As Integer,
                         lParam As Long, ReturnVal As Long)
    MsgBox "Just got a WM_NCHITTEST"
End Sub

Sub MsgBlaster2_Message (MsgVal As Integer, wParam As Integer,
                         lParam As Long, ReturnVal As Long)
    MsgBox "The message value was " + Str$(MsgVal)

End Sub

Sub MsgBlaster3_Message (MsgVal As Integer, wParam As Integer,
                         lParam As Long, ReturnVal As Long)
    Dim Msg As String
```

```
        If wParam = 0 Then
            Panel3d1.Caption = ""
        End If

        Msg$ = "Menu Option Currently is " + Str$(wParam - 1)
        Panel3d1.Caption = Msg$

    End Sub
```

Message-Handling Order

The Message Blaster can handle the messages it captures in three ways. The *MsgPassage* property array takes one of the following values to specify the message-handling order.

Value	Short name	Description
–1	PREPROCESS	Passes the message to the object's window procedure first, then triggers the Message Blaster's **Message** event.
0	EATMESSAGE	Triggers the Message Blaster's **Message** event without passing the message to the object's window procedure.
1	POSTPROCESS	Triggers the Message Blaster's **Message** event first, then passes the message to the object's window procedure. This is the default behavior.

Here is a **Form_Load** event procedure that uses the *MsgPassage* array:

```
Sub Form_Load ()
    Const PREPROCESS = -1
    Const EATMESSAGE = 0
    Const POSTPROCESS = 1

    ' The first blaster catches hit-test messages
    ' and uses the default preprocess passage.
    MsgBlaster1.hWndTarget = text1.hWnd
    MsgBlaster1.MsgList(0) = WM_NCHITTEST

    ' The second blaster catches left-button-down and right-button-down
    ' messages. Left-button-down messages are not sent to the control;
    ' right-button-down messages are.
    MsgBlaster2.hWndTarget = text2.hWnd
    MsgBlaster2.MsgList(0) = WM_LBUTTONDOWN
    MsgBlaster2.MsgPassage(0) = EATMESSAGE
    MsgBlaster2.MsgList(1) = WM_RBUTTONDOWN

    ' The third Message Blaster catches menu select messages
    ' for the example program.
    MsgBlaster3.hWndTarget = example.hWnd
    MsgBlaster3.MsgList(0) = WM_MENUSELECT
```

```
End Sub
```

Some Warnings

Message Blaster does, in some ways, subvert Visual Basic. Please be careful when you use this control; avoid using a Message Blaster for messages that Visual Basic already supports (for example, **Click** and **KeyPress**). Although Message Blaster may seem to work just fine, it may not always generate expected results. For example, if an application processes a button-up message and does not send it to the control, the control will look clicked, when it actually isn't.

You can have up to 25 Message Blasters on a system at one time. This limit covers all applications, not 25 Message Blasters per application. To reduce the number of Message Blasters that your application uses, you can reassign the target object for a Message Blaster at run time. For example, if your application has five edit controls and you want to process messages for each one, you can use one Message Blaster and change its target object as the focus moves to each edit control. This method also saves coding effort if each target object has the same or similar message-handling needs.

You can use the Message Blaster with Visual Basic version 1.0. However, this earlier version of Visual Basic requires more work because it does not provide the *hWnd* property for objects; you need to hunt down the object's window handle by using Windows functions such as **GetWindow** and **GetFocus**. If you're using Visual Basic 1.0, we strongly suggest that you upgrade to version 2.0 or later.

The Details of the VBX

This section describes how the Message Blaster control was implemented, for those of you who are interested in such things and who know how to develop VBX controls.

When an application that uses a Message Blaster is run, the application creates a Message Blaster control. When this happens, the Message Blaster finds a place for the control's information in its array of Message Blaster structures, initializes the data items, and sets up the About box for the control:

```
case WM_NCCREATE:
{
   int i;

   // Look for an open slot in the array and save the hctl.
   for(i = 0; i < MAX_MSGBLASTERS; i++)
   {
      if(controls[i].hctlMain == NULL)
      {
         controls[i].hctlMain = hctl;
         break;
      }
   }

   // If developers are trying to use too many blasters, they are
   // probably doing something wrong. Besides, using 25+ blasters
```

```
   // slows down the entire system.
   if (i == MAX_MSGBLASTERS)
   {
      MessageBox(hwnd, "Too many message blasters", "Error!", MB_OK);
      return(0);
   }

   // Make sure that the hWndTargets are zeroed out.
   LpblastDEREF(hctl)->hWndTarget = NULL;

   // Initialize the passage variable to pass messages to the original
   // procedure after firing the event. This is the default behavior.
   for(i = 0; i < MAX_MSGBLASTERS; i++)
      LpblastDEREF(hctl)->MsgPassage[i] = 1;

      VBSetControlProperty(hctl, IPROP_MSGBLASTER_ABOUTBOX,
      (LONG)(LPSTR)"Click on \"...\" for the About Box ---->");

}
break;
```

When Visual Basic sends a VBM_SETPROPERTY message for the IPROP_MSGBLASTER_HWNDTARGET property to the Message Blaster control, the Message Blaster subclasses the target object's window procedure:

```
case IPROP_MSGBLASTER_HWNDTARGET:
{
   int i;

   // We only process this message if we are in run mode.
   if (VBGetMode() != MODE_DESIGN)
   {

      // Find the next slot and save the hWndTarget.
      for(i = 0; i < MAX_MSGBLASTERS; i++)
      {
         if(controls[i].hctlMain == hctl)
         {
            // Check to see if this blaster is already being
            // used to subclass a control. If it is,
            // unsubclass it. This could happen if the
            // Visual Basic programmer changes the target
            // control at run time.
           if(controls[i].hWndTarget)
              UnSubClass(controls[i].hWndTarget,hctl);

           controls[i].hWndTarget = (HWND)lp;
           break;
         }
      }

         // Subclass the target control.
      if (!FSubClass((HWND)lp))
         MessageBox(hwnd, "Unable to subclass control", "Error", MB_OK);
```

```
      }
   }
   return 0L;
```

For a complete explanation of subclassing, see the "Safe Subclassing" technical article on the Microsoft Developer Network CD (look under Technical Articles, Windows Articles, Window Manager Articles). Here is how the Message Blaster subclasses its targets.

```
//----------------------------------------------------------------------------
// Aux fn to subclass only valid windows, and only when we're not
// already subclassing a window. Takes care of setting up and maintaining
// the two necessary globals. Returns TRUE if it actually is subclassed,
// FALSE otherwise.
//----------------------------------------------------------------------------
BOOL FSubClass(HWND hwnd)
{
   int i;
   HCTL hctlMain;

   if (!IsWindow(hwnd))
      return(FALSE);         // Invalid hwnd parameter.

   hctlMain = FindhctlMain(hwnd);
   if(!hctlMain)
      return(FALSE);


   if(i < MAX_MSGBLASTERS)
      if (LpblastDEREF(hctlMain)->lpfnOrigProc)
         return FALSE;      // Already subclassing.

   LpblastDEREF(hctlMain)->lpfnOrigProc =
      (FARPROC)SetWindowLong(hwnd, GWL_WNDPROC, (DWORD)SubClassProc);

    return TRUE;
}
```

Once subclassing is in place, each time Windows sends a message to the target window, the Message Blaster's **SubClassProc** procedure receives the message instead. If this message is among those that the Message Blaster control is monitoring, the Message Blaster handles the control's **Message** event and, depending on the *MsgPassage* value, passes the message to the target object's window procedure. If the message is not one of the monitored messages, the Message Blaster's subclass procedure passes the message directly to the target object's window procedure.

```
//----------------------------------------------------------------------------
// Subclass routine for target control window procedure.
//----------------------------------------------------------------------------
LONG FAR PASCAL _export SubClassProc(HWND hwnd, UINT msg, WPARAM wp, LPARAM lp)
{
   LONG rc;
   int  i, j;
   UINT m;
   HCTL hctlMain;
```

```
// Find the correct hctlMain.
// I could have called FindhctlMain, but I wanted to avoid the overhead.
// We need the most speed we can get in this routine.
for(j = 0; j < MAX_MSGBLASTERS; j++)
   if(controls[j].hWndTarget == hwnd)
      break;

// The first part of this is a sanity check.
if(j == MAX_MSGBLASTERS)
   MessageBox(hwnd, "Internal MsgBlaster Error", "NO BLASTER", MB_OK);
else
   hctlMain = controls[j].hctlMain;

for (i=0; i < MAX_TRAPPABLEMSGS -1; i++)
{

   // Get out of the loop if hctlMain is no longer valid.
   // Since you fire an event, it is possible for hctlMain to
   // disappear during that event, so you must check to see if hctlMain
   // is still non-NULL. This is also why I added the NULLing out of
   // this variable to the WM_DESTROY case.
   if (!hctlMain)
      break;

   // Firing an event invalidates pMsgBlaster!!!!
   // You MUST deref it each time through this loop!!!
   m = (UINT)LpblastDEREF(hctlMain)->MsgList[i];

   // Get out of the loop if no more messages.
   if (!m)
      break;

   if (msg == m)
   {
      int eat;

      // Now that we have a hit, determine if we should pass the
      // message on to the original procedure first, last, or
      // if we should eat the message entirely.
      eat = LpblastDEREF(hctlMain)->MsgPassage[i];
      switch(eat)
      {
      case -1: // Call the original procedure first.
         rc = CallWindowProc(LpblastDEREF(hctlMain)->lpfnOrigProc,
                             hwnd, msg, wp, lp);
          rc = FireMessage(hctlMain, msg, wp, lp, rc);
         if (msg == WM_DESTROY)
         {
            MessageBox(hwnd,
         "Can't call original proc with WM_DESTROY prior to firing event.",
         "Error", MB_OK);
            break;
        }
```

```
                return rc;

            case 0: // Eat the message.
                rc = FireMessage(hctlMain, msg, wp, lp, rc);
               if (msg == WM_DESTROY)
               {
                  MessageBox(hwnd,
                     "Can't eat WM_DESTROY message.",
                     "Error", MB_OK);
                  break;
               }
                rc;

            case 1: // Pass it on afterward.
                rc = FireMessage(hctlMain, msg, wp, lp, rc);
                break;

            default:
               MessageBox(hwnd,
                 "Default case in passage test. Should never get.",
                     "Error", MB_OK);
               break;
     }
  }
    }

    // Call the original window procedure.
   rc = CallWindowProc(LpblastDEREF(hctlMain)->lpfnOrigProc, hwnd, msg, wp, lp);

    // Strictly speaking, it is not necessary to unsubclass this window
    // since it is being destroyed, but UnSubClass does *need* to reset the
    // necessary global variables, and modular programming says we shouldn't
    // do that from here, when UnSubClass can do it for us.
    if (msg == WM_DESTROY)
       UnSubClass(hwnd, hctlMain);

    return rc;
}
```

A Warning About Changing MSGBLAST.VBX

If you need to change the Message Blaster custom control, please remember to change both the VBX name and its module name in the MSGBLAST.DEF file. Failure to do so may break other applications that use the Message Blaster and will make you a very unpopular developer.

Credits

Thanks to Chris Fraley, who helped Ed Staffin find a nasty bug, and to Jim Cash and Randall Kern for providing a nice improvement to the program.